Big Data intelligence Homework 2

Christoffer Brevik

November 26, 2024

Introduction

This report explains my findings and the results of my code, implemented to solve the homework regarding the study of personalized recommendation by leveraging collaborative filtering and matrix decomposition techniques on a dataset derived from the Netflix recommendation contest. The dataset contains user ratings for movies, where each user's rating is represented on a scale from 1 to 5. The goal is to predict missing ratings in a large user-item matrix, providing recommendations based on existing user preferences.

Note about the programs

My code uses the *Pandas* library to import and preprocess data, for matrix operations it uses the *numpy* library and for plotting it uses *pyplot*. The original code was a Jupyter file and the code has been exportet to .py afterwards

Some of my code was also altered to use *cupy* instead of *numpy*, so the programs could utilize `Google Colab's` GPU processing, which sped up the calculation time greatly. This is completely identical to the non-cuda code, with the exception of replacing all numpy arrays and operation with cuda variants. For **Task 3**, this is the code that gave me my optimal `k`, `alpha` and λ values. I did run the non-cuda version with these values to assure the codes were equal.

1 Data Pre-processing

This section describes how I preprocessed the data given in the assignment. **My final solution uses the whole dataset**. I will

1.1 Loading data into Dataframes

The first step is to load and preprocess the dataset, which consists of several files: users.txt, movie_titles.txt, netflix_train.txt, and netflix_test.txt. We begin by loading the data into pandas DataFrames for easier manipulation:

```
google_colab = False
local_root = 'Project2-data'
colab_root = '/content/drive/MyDrive/Colab Notebooks/Project2-data/'
root = colab_root if google_colab else local_root

# Load data
users = pd.read_csv(f'{root}/users.txt', sep=' ', names=['user_id'],
header=None)
movies = pd.read_csv(f'{root}/movie_titles.txt', names=['movie_id', 'year', 'title'],
header=None, on_bad_lines='skip')
movie_ids = movies['movie_id']

ratings_train = pd.read_csv(f'{root}/netflix_train.txt', sep=' ',
names=['user_id', 'movie_id', 'rating', 'date'], header=None)
ratings_test = pd.read_csv(f'{root}/netflix_test.txt', sep=' ',
names=['user_id', 'movie_id', 'rating', 'date'], header=None)
```

1.2 Creating a User-Movie reivew matrix

Next, we create a user-item rating matrix from the training set, where each row represents a user and each column represents a movie. The values in the matrix are the reviews given by use x for movie y. Missing values are filled with zeros.

```
# Create matrix
matrix = ratings_train.pivot(index='user_id', columns='movie_id', values='rating').fillna(0)

# Convert to numpy matrix for computation
vector_matrix = matrix.values
```

1.3 Matrixes to easily get locations of movies and users

We also create mappings from user IDs and movie IDs to matrix indices for efficient lookup during computations. I requited these as not every user id was in the dataset, so I couldn't actually map the user id's to the location in the User-Movie review matrix.

```
# Map user and movie IDs to matrix indices
user_id_to_index = {user_id: idx for idx, user_id in enumerate(matrix.index)}
movie_id_to_index = {movie_id: idx for idx, movie_id in enumerate(matrix.columns)}
```

2 User-based Collaborative Filtering Implementation

In this assignment, we implemented a user-based collaborative filtering (UserCF) algorithm for movie recommendations. The goal was to predict movie ratings based on the ratings of similar users and evaluate the performance of the model using Root Mean Squared Error (RMSE).

Algorithm Overview

The implemented algorithm consists of two main components: the computation of user similarities and the prediction of movie ratings.

Computation of User Similarities:

The user similarity matrix is computed using cosine similarity, as described in the assignment, where the cosine similarity between two users u_i and u_j is defined as:

$$\text{similarity}(u_i, u_j) = \frac{\mathbf{r}_{u_i} \cdot \mathbf{r}_{u_j}}{\|\mathbf{r}_{u_i}\| \|\mathbf{r}_{u_j}\|}$$

where $\mathbf{r}_{u_i}$ and $\mathbf{r}_{u_j}$ are the rating vectors of users u_i and u_j, and $\|\mathbf{r}\|$ denotes the Euclidean norm of the vector.

Rating Prediction:

For each target user, we predict the rating for a specific movie by taking a weighted sum of the ratings given by similar users:

$$\hat{r}_{ui} = \frac{\sum_{v \in \text{neighbors}(u)} \text{similarity}(u, v) \cdot r_{vi}}{\sum_{v \in \text{neighbors}(u)} |\text{similarity}(u, v)|}$$

where r_{vi} is the rating given by user v to movie i, and neighbors(u) refers to the set of users most similar to the target user u.

The prediction is restricted to users who have already rated the movie, avoiding the consideration of unrated items.

2.1 Matrix Operations and Optimization

Given the sparsity of the user-item matrix, matrix operations were used to implement the algorithm efficiently. This approach leverages optimized linear algebra functions like matrix multiplication and normalization using NumPy.
This vectorized approach avoids the need for for-loops, which are computationally expensive, especially for large datasets. The result is a much more efficient algorithm that can handle large, sparse matrices typical of collaborative filtering tasks.
The only exception to this is the `RMSE` where I was unable to implement a fully vectorized solution, resulting in one for-loop. Still I've tried to optimize the alrogithm as much as possible.

2.2 Performance Evaluation

The performance of the algorithm was evaluated using RMSE, computed as follows:

$$\text{RMSE} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (\hat{r}_{ui} - r_{ui})^2}$$

where $\hat{r}_{ui}$ is the predicted rating for user u and movie i, and r_{ui} is the actual rating.

2.3 Results:

The RMSE was computed using the test dataset, which contains user-movie ratings that were not used during the training phase. The algorithm achieved an RMSE value of $\sim$**0.988** on the test set.

2.4 Code Implementation

I broke my code implementation into several key functions:

- `compute_user_similarities(matrix)`: Computes the user similarity matrix using cosine similarity.
- `userCF(matrix, user_id, movie_id, similarity_matrix)`: Predicts the rating for a given user-movie pair using the similarity matrix.
- `userCF_rmse(similarity_matrix, matrix, ratings_test)`: Computes the RMSE of the model on the test dataset.
- `compare_predictions(predictions, similarity_matrix)`: Compares predicted ratings to actual ratings for a set of randomly selected user-movie pairs.

The following snippet illustrates the process of computing the user similarity matrix and evaluating RMSE:

```
print("Computing user similarities")
similarity_matrix = compute_user_similarities(matrix)
print("Done computing similarities :) \n")

# Evaluate RMSE for a specific test set size
print("Computing RMSE")
test_rmse = userCF_rmse(similarity_matrix=similarity_matrix, matrix=matrix, ratings_test=ratings_test
print(f"Test RMSE: {test_rmse}")
```

2.5 Time Consumption

Running my code takes approximately 4 minutes. This is with caching results, meaning that results can be reused, greatly reducing runtime

2.6 Conclusion

The user-based collaborative filtering algorithm successfully predicted movie ratings using a similarity-based approach. The results were evaluated using RMSE, and the performance of the model was satisfactory. Future improvements could include incorporating additional techniques such as regularization or experimenting with different similarity metrics to further enhance the model's accuracy.

3 Matrix Decomposition Algorithm

For this assignment, I implemented a matrix decomposition algorithm with gradient descent.

a: Changes in the Target Function and RMSE for $k = 50$, $\lambda = 0.01$

For the case of $k = 50$ and $\lambda = 0.01$, the matrix decomposition algorithm was executed over multiple iterations, with the target function value (loss) and the RMSE on the test set tracked at each iteration. Below is a summary of the observations and results. Here, both increased to a point so high that my algorithm stopped itself.
Below are the plots illustrating these observations:

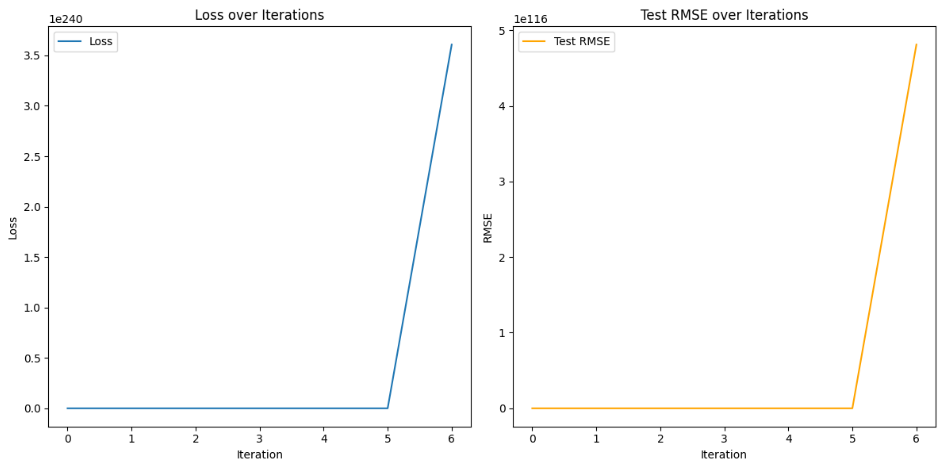

Figure 1: Target Function Value vs. Iterations. The plot shows a great increasing trend in the target function value across iterations, converging after approximately *¡number¿* iterations.

This shows that I needed more fine tuned values to solve the task, and showed how volatile this alogirthm can be. The final RMSE is therefore **Infinite** for these values

3.0.1 (b) Optimal Values for α, k, and λ

In this part of the assignment, I explored various combinations of the hyperparameters α (learning rate), k (number of latent factors), and λ (regularization parameter) to identify the optimal set of values that minimize RMSE on the test set. After extensive experimentation with different configurations, the following combination was found to yield the best performance:

- $k = 10$
- $\alpha = 5 \times 10^{-5}$
- $\lambda = 1$
- `max_iter` $= 10{,}000$

Using these parameters, the matrix decomposition algorithm achieved a final RMSE of 0.8111 on the test set. This result demonstrates the importance of careful tuning of hyperparameters to balance model complexity, learning stability, and generalization performance.
The choice of $k = 10$ indicates that a relatively small number of latent factors sufficed to capture the underlying patterns in the dataset, while $\alpha = 5 \times 10^{-5}$ ensured stable convergence during gradient descent. A regularization parameter $\lambda = 1$ struck the right balance between underfitting and overfitting, reducing the impact of over-penalizing larger weights.

4 Comparing CF and Gradient Descent

4.1 Collaborative Filtering

Collaborative filtering is faster to calculate and simpler to implement, making it a practical choice for systems that require straightforward recommendations with minimal computational overhead. It is particularly effective in scenarios with dense interaction data. However, it struggles with sparse datasets, cold start problems, and tends to favor popular items, which can reduce recommendation diversity.

4.2 Gradient Descent

Gradient descent offers higher accuracy by uncovering latent patterns in the data through optimization techniques. It is well-suited for sparse datasets and can incorporate additional constraints or side information. However, it is more complex to implement and highly sensitive to input values, requiring careful fine-tuning of parameters such as the learning rate, regularization, and the number of latent factors. This volatility can make the training process more time-consuming and computationally expensive. Still it yielded clearly better results, and I believe a RMSE of below 0.8 is very much possible if you have the right values